2012

Addressing the feasibility of USI-based threads scheduler on polymorphic computing system

Zhang Zhang
Iowa State University

Addressing the feasibility of USI-based threads scheduler on polymorphic computing system

by

Zhang Zhang

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Akhilesh Tyagi, Major Professor
Joseph Zambreno
Thomas Daniels

Iowa State University
Ames, Iowa
2013

DEDICATION

I would like to dedicate this thesis to my mother, who raises me along and supports my study these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects during my study at Iowa State University. First and foremost, I would like to express my thanks to Dr. Akhilesh Tyagi for his guidance, patience and support throughout my research and the writing of this thesis. His wisdom and insights are the most important compass of my work. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Joseph Zambreno and Dr. Thomas Daniels. I would additionally like to thank Yanan Cao for his help in preparation of the necessary forms of this thesis.

ABSTRACT

The consistent advances in IC technology result in ever increasing number of transistors. There is more and more interest attracted on the issue of using these transistors in computing more efficiently. The CMP (Chip Multi-processors) is predicted to be one of the most promising solutions for this problem in future. The heterogeneous CMP is supposed to provide more computing efficiency compared to the homogeneous CMP architecture; but it requires complex processing art for manufacturing, which makes it less competitive in the old era. Nowadays, the complicate SOC(System On Chip) manufacturing techniques are pacing fast. This is leading us inexorably to heterogeneous CMP with diverse computing style resources like general purpose CPU, GPU, FPGA, and ASIC cores. In the heterogeneous CMP architecture, the generous purpose CPU provides coverage for almost every type of computing, while the non von-Neumann cores harvest energy and processing time for specific computing. The polymorphic system is defined as a heterogeneous system that enable a computing thread to be dynamically selected and mapped to multiple kinds of cores. A polymorphic thread is compiled for multiple morphisms afforded by these diverse cores. The resulting polymorphic computing systems solve two problems. **(1)** Polymorphic threads enable more complex, dynamic trade-offs between delay and power consumption. A piecewise cobbling of multiple morphism energy-delay profiles offers a richer Energy-Delay(ED) profile for the entire application. This in turn helps scale the proverbial ITRS "red-brick power wall". **(2)** The OS scheduler not only picks a thread to run, it also chooses its morphism. Previously, the scientists and engineers prefer using the numerical E·T results to evaluate the design trade-offs, which is challenged to not fit on the future mobile systems design in this thesis. In the mobile systems, whose primary role is "enhanced terminals" - user interface to cloud hosted computing backbone, user satisfaction ought to be the primary goal. We propose a scheduler to target User Satisfaction Index (USI) functions. In this thesis, we develop a model for a mobile polymorphic embedded system. This model

primarily abstracts the queuing process of the threads in the OS operation. We integrate a polymorphic scheduler in this model to assess the application design space offered by polymorphic computing. We explore several greedy versions of a polymorphic scheduler to improve the user satisfaction driven QoS. We build a polymorphic system simulation platform based on SystemC to validate our theoretical analysis of a polymorphic system. We evaluate our polymorphic scheduler on a variety of application mix with various metrics. We further discuss the feasibility of USI-based polymorphic scheduler by identifying its strengths and weaknesses in relation to the application design space based on the simulation results.

CHAPTER 1. INTRODUCTION

1.1 Motivation

Mobile systems serve as display terminals of the old days. Both energy and power are at a premium in these portable systems. Maximizing the battery life is the holy grail of performance metrics for mobile systems. However, computational demands made on these platforms are non-trivial. An aggressive Energy-Delay profile based mobile chip-set will exhaust the battery quickly even when non-delay-critical applications are active. This dictates architecture level support for energy-aggressive energy-time profiles. But the need for high performance versus low energy Energy-Delay profile is dynamic and context dependent. Multiple energy-time profiles are needed. The solution is to include Energy-Delay profile diverse cores in a heterogeneous multi-core mobile chip-set. These cores include software morphism general purpose CPU (perhaps both scalar and superscalar), FPGA based reconfigurable core, a GPGPU, and application-specific integrated circuit (ASIC) core. Software for heterogeneous systems may contain polymorphic threads - threads compiled with multiple morphisms with identical behavior - such as CPU morphism and FPGA morphism. Such a diverse Energy-Delay profile cores and threads based system is called a polymorphic computing system. The Energy-Delay profile of each morphism for the identical behavior prescribed by a thread differs offering newer flexibility to an application developer.

One can visualize feasible Energy-Delay space for a given application. The upper envelope is defined by the CPU morphism for each thread. The lower envelope is defined by either ASIC core ET profile or FPGA core ET profile for all the threads. A more practical lower envelope may only consider ET profile of the supported morphisms for each thread.

User interaction is tricky since human beings are impatient by nature. This leads to an

interesting opportunity for optimization. The classical process/thread scheduling endeavors to maximize performance or some other explicit physical performance related attribute such as energy or throughput. However, if the objective is user satisfaction, that too, with mostly user interface computing, a different optimization objective might be in order. We propose user satisfaction index (USI) as a function with the following characteristics. It is a saturating function of classical performance metrics. Making something faster indefinitely does not lead to increasing user satisfaction. User satisfaction and dis-satisfaction saturate at some point. In this paper, we develop the USI concept further, and assess systems built with it.

Need for such polymorphic systems is likely to grow based on the following trends. We can foresee that the computation throughput required by the mobile applications in the future will keep increasing [16]. The major design tradeoffs have already shifted from the traditional *speed versus area* to *speed versus energy* [8] in the current embedded CPU design. This trend will dominate in the near future. The heterogeneous and homogeneous chip-multi-processors (CMP-s) are two of the candidates proposed for solving the problem of balancing energy consumption and processing speed [8]. While the hardware techniques for attaining a single Energy-Delay profile are well developed and their potential improvements have already been assessed in the literature such as [8, 11], the software techniques are still under-explored and under-utilized. The use of OS and application architecture to patch together a desirable Energy-Delay profile from many provided by diverse hardware cores has also not been explored. We believe that the software solutions, like smarter polymorphic scheduler [14], tailored compiler for reconfigurable computing [4] or even specialized programming language [7] will further enhance the user experience in the next-generation mobile computing system. Our research in this paper is mainly about building a novel model for a polymorphic scheduler for an embedded CMP architecture and the potential performance improvements achievable from polymorphic computing.

Current Operating Systems (OSs) on mobile device like Android and iOS inherit many features from desktop computing era. Android is based on Linux which incorporates the following two major schedulers: **(1)** Completely Fair Scheduler (CFS) and **(2)** BFS [24]. According to [24], BFS is better for interactive tasks than CFS while the CFS outperforms the BFS for batch processing. Besides, CFS is especially suitable for multi-cores systems with large

number of cores, like 1024 cores. We should notice that both of these two schedulers follow the throughput focus of desktop computing. Neither of them considers the energy efficiency issue. The main-stream Android OS adopts the $O(1)$ scheduler. It adds some Real-Time (RT) scheduling solutions for certain tasks like video and audio applications. Bower *et al.* [3] have already demonstrated that the embedded processors with heterogeneous cores provide margins for the scheduler to improve the energy efficiency to prolong the battery life. The current scheduling techniques ignore these axes of application design space. They do not fit future mobile platforms well.

Our previous work [14] has presented a novel threads scheduler optimizing an USI based objective function. This scheduler picks a morphism for each scheduled thread as well. The overall goal is to maximize the USI (user satisfaction) within the resource constraints. This polymorphic scheduler is effective at USI optimization which indirectly balances the tradeoff between *speed and energy*. But there still are many unanswered questions about scheduling in a mobile, polymorphic heterogeneous computing system. **(1)** Is there a tight theoretical upper bound on the additional performance gain offered by polymorphic computing? **(2)** What kind of space - application mix and characteristics - is ideal or best terrain for polymorphic computing? Does polymorphic computing offer an advantage for the typical mobile embedded environments? **(3)** What kind of function should we adopt to model the user satisfaction and how to evaluate the USI-based scheduling? **(4)** Last but not least, does the USI-based scheduler really exploit the potential of polymorphic computing? To address these questions, **(1)** we have built a queuing model for a theoretical analysis of polymorphic systems and **(2)** we have built a System-C [10] simulator for simulating a heterogeneous, polymorphic computing system at an abstract level. We present different test scenarios and workloads to demonstrate the validity of our theory. We develop a model of traditional scheduling approach for a polymorphic computing platform as the baseline for evaluating the USI-based scheduler.

1.2 Background

1.2.1 Heterogeneous Computing and Polymorphic Computing

An example of a heterogeneous computing system is shown in Figure 1.1. There are various kinds of computation resources available: general-purpose CPU cores, FPGA cores, GPU cores and ASIC cores. The number and size (parametrization) of various computing cores/resources should be properly calibrated. Once chip is fabricated, these parameters and characteristics become immutable. Figure 1.1 includes routers and I/O controllers. The network-on-chip (NoC) is needed to make the computing cores virtual for a dynamic scheduling environment. I/O controller cores coordinate the data/instructions flows between the computing micro-cores and the memory. Note from Figure 1.1 that this system is based on the mesh network on chip (NoC) architecture[15]. A heterogeneous computing system can adopt any other NoC architecture, such as star, ring, or other architectures [19]. The NoC connection architecture would mainly affect the communication efficiency between the cores. We will not focus on this topic in our work and assume the communication channels between the cores to be ideal channels. In current practical heterogeneous computing systems, the software threads are mainly mapped to computation resources statically. The FFT computing will always be mapped to the FFT cores, even if the FFT cores are all occupied and threads must wait. Chung [6] argues for static mapping claiming that mapping computation to unsuitable resources will waste energy and prolong the processing time by one or two orders of magnitude. Another advantage of static mapping scheme is that it saves storage space compared to a dynamic mapping scheme. In order to support mapping one thread to two or more different architectures, we need to double or triple the storage resources to store different software images. This is assuming we do not deploy other software solutions that reuse the same software image. In architecturally diverse cores, sharing of the same binary image gets more and more difficult and challenging. We will call a heterogeneous computing system a polymorphic computing system if this system can map a thread to multiple cores with diverse architecture.

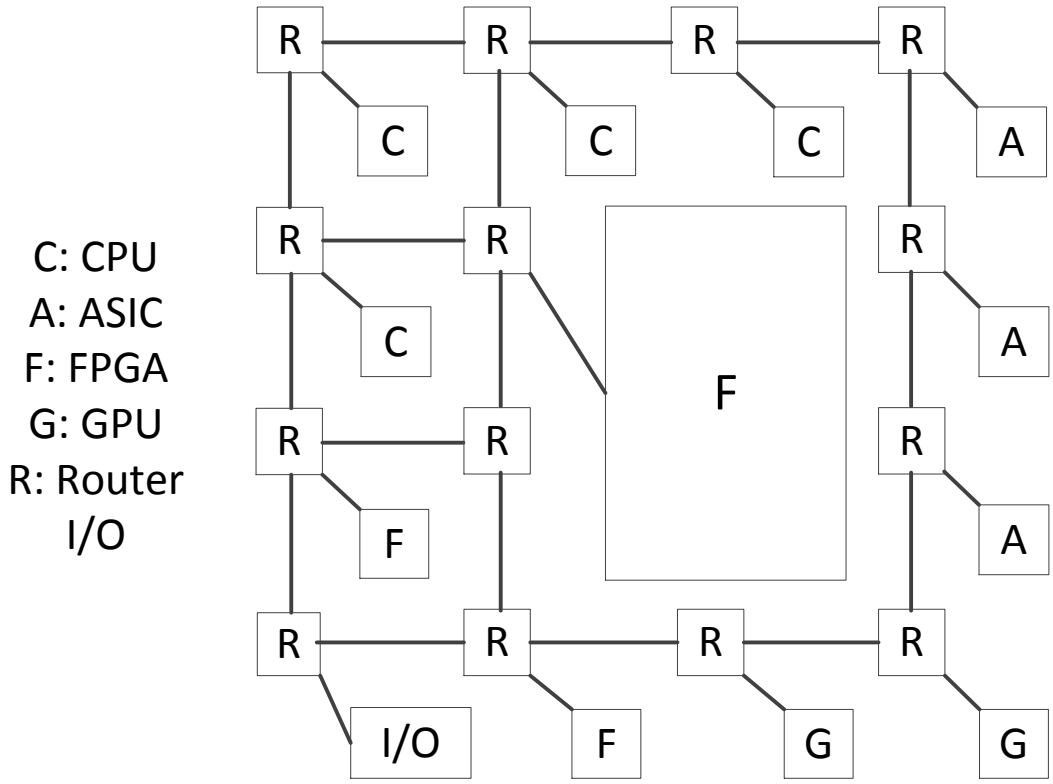

Figure 1.1 An Example of a Heterogeneous Computing System

1.2.2 The USI-based Scheduling on Polymorphic Architecture

In the preceding discussion, fixed static mapping of threads to cores or core types has the advantage of minimum energy and processing time, which favors the user satisfaction. This assumes that the static mapping is to the single core type with the minimum energy and delay. But this scheme ignores one important component of turnaround time which will greatly affect the user's feeling - the waiting time. The waiting time of each thread will increase significantly if many same type threads wait in the single core type ready queue. The scenario that many similar threads congest a single core type waiting queue is not rare in real software. Real program do not distribute their threads across all core type queues uniformly which would be the ideal (theoretical) model. Even a distribution of threads across multiple core type queues derived from some benchmark application suite does not/may not match the

distribution observed in real workloads. A computing system designed to be balanced with respect to benchmark workloads will fail to meet the timing and energy constraints under specific scenarios. A dynamic adaptive thread to core mapping overcomes such issues. One such greedy heuristic is when threads are adaptively moved to empty or low-occupancy queues. The scheduler sacrifices the processing time and energy for a shorter waiting time.

How should such a dynamic scheduling policy be evaluated? Decision to move a thread has iterative affects on waiting times of many other threads. Invariably, dynamic thread movement worsens Energy-Delay profile of the thread. A thread waiting in the FPGA core type queue, dynamically moved to CPU core type queue will have 3 times worse energy and 3 times higher time resulting in 9 times worse Energy-Delay profile. This means extra processing delay due to worse Energy-Delay profile; hence the expected waiting time savings for all the current threads waiting in the queue must be higher than the extra processing delay to justify the dynamic thread move. Otherwise it will not bring any benefits. If dynamic change in one thread's morphism will make turnaround time for all the current waiting threads shorter, then how to quantify the trade-off between this decision's extra energy consumption with the overall delay's reduction is a tricky problem. Our method for this is to consider the effect on USI Index from both the reduced delay and wasted energy. If a thread is waiting for a while but is not urgent (not on critical path) and the current battery level is low, then moving it to another morphism, which reduces the waiting time for this thread and other threads behind it in the queue, still may not be a wise decision. This is because any additional energy consumption in a low battery state may lead to significant degradation of USI. With a high battery level, the user may not focus on the battery at all and may prefer the shortest turnaround time. In such a system state, reduction in the waiting time for each thread may improve the USI enough to offset the USI degradation due to extra energy consumption in the worse Energy-Delay profile of the new morphism. We weight energy and delay with different parameters within the USI equation/model. Such a USI driven technique has potential to achieve an elegant balance between *speed and energy* based on the user satisfaction. This may lead to a new scheduling policy paradigm.

1.3 Related Work

1.3.1 The Extended Amdahl's Law

There are several papers discussing the extended Amdahl's laws for the homogeneous and heterogeneous computing systems. Hill & Marty [13] discuss a series of mathematical models based on the Amdahl's Law and Pollack's rules [2]. Symmetric CMP, asymmetric CMP and dynamic CMP models are analyzed. However, only the traditional von-Neumann cores are considered. The modeled performance metric is still the classical speed-up. Cassidy and Andreou [5] further extended the Amdahl's Law to include energy in its objective function. The objective function used in [5] takes weighted average of delay and energy with an arbitrary selection of weights. Paul and Meyer [20] revisit the Amdahl's law for heterogeneous CMPs. Their main lesson is that un-conventional computing resources, like DSP and ASIC will provide additional speed-up for the overall system.

Moncrieff *et al.* [17] revisited the Amdahl Law for the heterogeneous computing platform with another approach. The authors calculated out several scenarios using numbers derived from the numerical model based on the revisited Amdahl law. A positive evaluation of the heterogeneous architectures is given based on the numerical results. All these results have two common features; one is that they all use simple performance metrics to evaluate the system, like processing throughput, or the average energy consumption per unit computation. The other common feature is that they only consider static mapping of the tasks or threads to the computation resources, and thus ignore the potential of dynamic resource allocation for applications by the OS.

1.3.2 Performance Model of Heterogeneous CMP

The current IC technology enables us to fabricate cores with diverse computation architecture (CPU, FPGA, GPGPU, and ASIC units) into one chip, known as heterogeneous CMP. Many studies have been conducted to determine the computational performance of each component of the CMP within the heterogeneous architecture. Chung *et al.* [6] presents the comparative relationship between the conventional computational resources, the general pur-

pose CPU and unconventional cores (u-cores) like FPGA, GPU, and ASIC. Chung [6] estimates the speed advantage of FPGA over a CPU to be a factor of three on average which includes the reconfiguration time over some average temporal locality of behaviors. The FPGA core also has a factor of three advantage in power over a unit CPU core (a MIPS like scalar core). This means the energy advantage of FPGA core is a factor of 9 over a scalar CPU core. Similarly, Chung's simulation results indicate the ASIC core's energy advantage to be a factor of 100 over a CPU core. Although Chung's paper establishes the FPGA core's speed and energy advantage over CPU, DeHon's paper [9] presents the strong and weak points for both FPGA and CPU based on computational density metric. It points out that Chung's conclusions are not universal for all scenarios. DeHon shows that FPGA can outperform the CPU for special tasks that exhibit control parallelism. The explicit control of FPGA model wastes energy and reduces the system throughput for inherently sequential computation, like vector arithmetic add. Combining the results from [6, 9], we can generally model and tag the tasks/applications and threads to be CPU or FPGA type. Such a model can also determine the most suitable computation resources. This model can be applied with proper numerical parameters within a simulating environment.

1.3.3 Modeling the User Satisfaction Index

Shye *et al.* [22] presents a psychology experiment to explore the relationship between a user's satisfaction and computing system's performance metrics-speed. In this preliminary research, results revealed that the user's satisfaction of a service may not be linear in time such as frames per second (fps). Many other functions like step function, staircase function and constant function are possible. These results match our common sense. Imagine a video game application. If the processor provides a service at 20 fps, then the user will experience lag leading to dis-satisfaction. A service at 60 fps will be fast and satisfying enough for all users. It is expected that higher computing performance throughput requires higher energy. Satisfaction as a human sentiment tends to saturate. The service with 100 fps does not significantly alter a user's satisfaction compared with 60 fps service. This results in the additional energy resources (compared with the 60 fps service) wasted with no appreciable gain the main system metric of

user satisfaction. In an embedded game system powered by battery, this waste will reduce the battery life, which may lead to additional user dis-satisfaction. In fact, the user satisfaction seems to depend on the battery level state. Base on this intuitive idea, our previous work [14] proposed an USI-based scheduler to achieve better balance of speed and energy. The sigmoid function [18] for modeling user satisfaction was shown to be a viable approach. The sigmoid function's efficiency at modeling the variance of user-satisfaction [23, 26] was further examined. In our previous work, the scheduler's objective function focuses on the marginal USI gain per unit of energy. In this paper, we will further explore other possible forms and variations for user satisfaction model. We also argue that quadratic-form USI functions are well-suited for modeling delay based user satisfaction.

1.4 Organization of this Thesis

The rest of this thesis is organized as follows. The Chapter 2 presents our theoretical analysis of the polymorphic computing system. A basic queuing model and its extension for simulation are presented in the Chapter 2.1. A primitive theoretical analysis based on this model is also presented in this chapter. Chapter 2.2 discusses the USI functions and presents a greedy USI-based polymorphic scheduling algorithm. The simulation methodology and experimental results are demonstrated in Chapter 3. The Chapter 3.1 further introduces the details of the model and feature of the testbench. The Chapter 3.2 presents the numerical results in tables and histograms. Implications of the primary experiments' results are also discussed in this chapter. Chapter 4 offers our final conclusions. And we also give our opinions about this research's future topic in the Chapter 4.

CHAPTER 2. THEORETICAL ANALYSIS

2.1 A Theoretical Model

The OS scheduling is a classic queueing problem, and we can apply the well developed queueing theory to designing of the traditional OS scheduler. But there are several new features appear in the scheduler of polymorphic computing, thus we need new model to study the scheduling problem on polymorphic system. Firstly, the scheduler not only chooses the order of threads' execution , but also chooses each thread's type of morphism. This involves trade-off of processing time and battery consumption, which is not modeled in the traditional OS queueing theory. Secondly, the optimization targets of the scheduling algorithm shift from traditional metrics like queues' length, average waiting time/turnaround time, or cores' load balance to the new USI index. We introduce the USI-function, which is a non-linear function of delay and energy. Thirdly, introducing the model of battery means that the system is not a time-invariable system. Then the old random process analysis method like the Markov-Chain approach won't fit for this model. Due to the reasons listed above, we build our model embedded with the new features of the polymorphic computing system. We should notice that since the model is based on a time-variable system, little analytical results can be obtained from this model, and we have to mainly rely on the numerical calculation based simulation to study the scheduling on polymorphic computing.

2.1.1 Definition of Threads' Type

In the original Amdahl's Law, the computing workloads are modeled with two characteristics: "the portion that can be improved" and the "portion that cannot be improved". Amdahl law points out that the "un-improvable portion" would be the bottleneck for improving the sys-

tem's overall performance. In multi-core systems, an application can be executed on multiple cores in parallel as the "improvable portion". In heterogeneous computing platforms, we will change this definition because not every kind of computing can be mapped to all cores. CPU and FPGA core types can host almost any kind of computing thread. But DeHon's paper[9] indicates that some types of computing (like vector add) should not be mapped to FPGA tiles when vector length (indicating degree of sequentiality) is long. From their results, we can assert that CPU cores can host every type of computing; while the FPGA cores can host a large fraction of computing threads; leaving some computing only to the CPU cores. Similarly, we can assert that GPU cores can host some specific data-parallel tasks, and the ASIC can only be used for a small fraction of computing threads. In our model of heterogeneous computing systems, the computing will be separated into different types. Each type accounts for fraction f_1, $f_2, \ldots, f_n$ of the total threads for the core type 1, 2, $\ldots, n$ respectively. The Venn diagram in Figure 2.1 demonstrates the coverage of different different computing types.

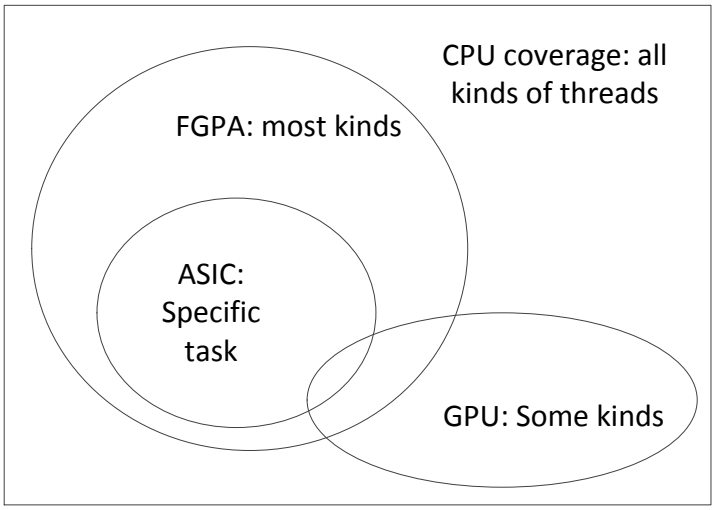

Figure 2.1 Coverage of Different Computing Types

2.1.2 The Model of Flow-Mapping

We propose a simple queuing model for abstracting the problem of thread mapping and scheduling in a heterogeneous computing platform. In this model, the waiting queue (which is common among different OSs) is modeled as "water tank"; the threads are generated by generators which are equivalent to the water flow's source in the model; and the computing resources are modeled as the water flow's drains. The mapping procedure controlled by the scheduler (modeled as a water valve) is the "flow" (Figure 2.2). Figure 2.2 (a) presents the scheduler abstraction. In this model, we assume that the scheduler knows the exact generation rate of different thread types. We also assume there is no waiting queue for threads. We will work out a formula to indicate the scenarios that harvest additional USI through polymorphic computing based morphism control with scheduler. Figure 2.2 (b) presents the more refined simulation model of the scheduler abstraction in Figure 2.2 (a). In real practical OSs, the scheduler will not get accurate information on how many threads of various types will arrive in the future. We need some queues to hold the waiting threads. In Figure 2.2 (b) model, we assume that the scheduler cannot know the exact incoming rate, but can observe the height of the tanks to infer the generation rates. Some practical formulas using this simulation model are also deduced.

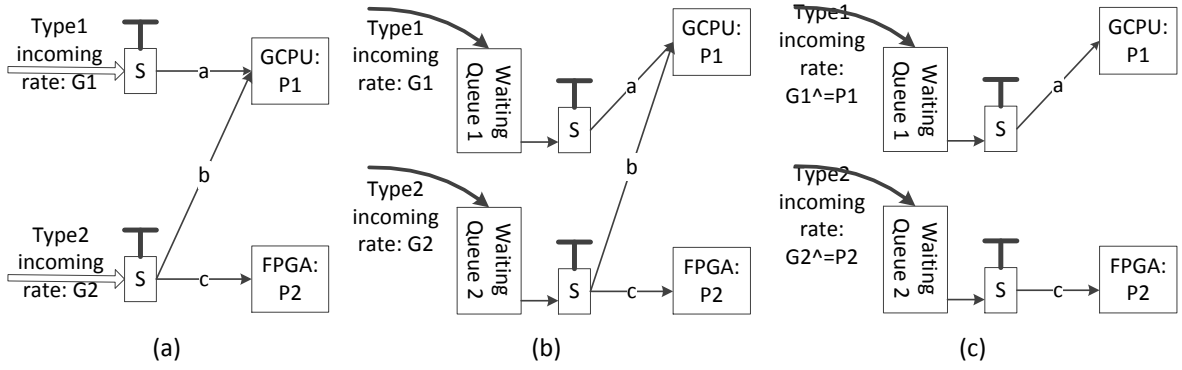

Figure 2.2 The Flow-Mapping Model

Figure 2.2 (c) presents the static design of a polymorphic computing system. We define the expected generation rates of source (generator) 1, 2 as $\hat{G_1}$, $\hat{G_2}$ respectively. The maximum

processing throughput of the CPU cores (over all CPU cores) and FPGA tiles (over all FPGA cores) are defined as P_1, P_2 respectively. In the static design of the system, the optimal solution is $\hat{G_1} = P_1$ and $\hat{G_2} = P_2$. Under this design, static b flow (from Type-2 to Type-1) should not exist. Chung [6] concludes that a static b flow from Type-2 to Type-1 morphism will waste power by leaving the better Energy-Delay profile of Type-2 on the table and introduce extra delay to both queues. During system chip design, the computing architecture type resources (Type-1 and 2 in this example) should be set in the ratio of expected application mix of the two types so as to eliminate the static b flow. A dynamic b flow is necessary only when Type-a (Type-1 thread to Type-1 CPU core) and Type-c flows (Type-2 thread to Type-2 FPGA core) are not balanced (generation rate of that type does not match the processing rate). We define the instantaneous generation rates of source of Type-1,2 as $G_1 = \hat{G_1} \cdot \lambda_1$, $G_2 = \hat{G_2} \cdot \lambda_2$ respectively. The λ_1 and λ_2 are the out of balance coefficients of the each source type. In classical queuing models, the source generates the threads at time intervals specified by a random distribution, like Poisson distribution or Normal distribution. But this generalized mathematical model is far from real scenarios. In real scenarios, there is a significant probability that during a long period of time, a specific thread type generation rate is extremely high or low. Here we use the "unbalance coefficient" (out of balance) to model the transient nature of the generation rate. The unbalance coefficients are from the application characteristics. In the theoretical model of Figure 2.2 (a), the flows a and b, are two thread flows mapped by the scheduler. Due to type restrictions, the threads generated by generator-1 cannot be mapped to an FPGA core, but the threads of Type-2 can be mapped to both CPU and FPGA cores. The transient variations make the static resource allocation during the design of a polymorphic system difficult. Let λ_{1max} and λ_{2max} be the maximum transient (unbalance) variation observed in the generation rate of Type-1 and Type-2 threads. If we are willing to allocate redundant resources that will idle frequently, we would set (design for) $P_1 = \hat{G_1} \cdot \lambda_{1max}$, $P_2 = \hat{G_2} \cdot \lambda_{2max}$ at the chip design stage. This setting is impractical in an embedded system. In an embedded computing platform, the area constraints, power constraints, bandwidth constraints, and other constraints are tight. Hence, designing an embedded system with computing resources to match the maximum variation in threads' transient generation rate is next to impossible. In this paper, we would assume that $G > P$

only for very short periods. When the dynamic G is higher than static P for long periods, the buffers of the computing system will overflow, and some feedback scheme would throttle the threads generation rate. Consider a user with a mobile device like iPad. If the device freezes, becomes non-responsive due to overwhelmed resources, the users cease to generate new activity. The user serves as a negative feedback loop. Modeling this feedback is hard both in mathematical analysis and for numerical simulation. Hence we will assume that maximum dynamic $G \leq P$.

2.1.3 Different Operating Zones

In the preceding discussion, we have defined the basic elements of the queuing model. One key component that remains un-discussed in this model is the scheduler, the valve of the water flow paths. In the model presented in Figure 2.2 (a), we assume that the scheduler knows the exact thread generation rates of each type. The schedulers with different scheduling policy will adjust the valve under different scenarios. The speed-oriental scheduler, like common Android scheduler, would issue as many threads as possible to maximize the throughput. It improves the overall average delay for the applications. This kind of scheduler would keep b-flow open/going as long as $\lambda_2 > \lambda_1$. This scheduler would stop increasing b flow only when $(a+b)/c = P_1/P_2$. Here, a, b, and c denote the task flow rate through links a, b, and c respectively. On the other hand, an energy-focused scheduler would likely shut down the b flow. This is because mapping Type-2 threads to FPGA cores will always save energy. This kind of schedulers will ignore the extra waiting time of Type-2 threads with the primary focus on prolonging the battery life. The USI-function based scheduler would adjust the b flow between the speed scheduler and power scheduler listed above. Since the energy-scheduler and speed-scheduler define the boundaries of the USI-balanced scheduler, we can determine the USI-scheduler's flow b's viable zone, which indicates the viable operating zone for the USI-based scheduler. The formula(2.1) defines b's boundary conditions:

$$\begin{cases} a = \hat{G_1} \cdot \lambda_1, & b + c = \hat{G_2} \cdot \lambda_2; \\ a + b < P_1; & c < P_2; \\ (a+b)/P_1 < c/P_2; & \end{cases} \tag{2.1}$$

By solving these boundary conditions above, we can get the viable operating zone for b:

$$0 < b < \frac{P_1 \cdot \hat{G_2} \cdot \lambda_2 - P_2 \cdot \hat{G_1} \cdot \lambda_1}{P_1 + P_2} \tag{2.2}$$

Equation 2.2 indicates that the USI-based scheduler's ability to find a viable zone strongly depends on the flow source generation rate and other characteristics. If the work loads' parameters satisfy the condition $P_1 \cdot \hat{G_2} \cdot \lambda_2 < P_2 \cdot \hat{G_1} \cdot \lambda_1$, then the constrained USI-based scheduler cannot adjust the b flow at all. On the other hand, if $P_1 \cdot \hat{G_2} \cdot \lambda_2 >> P_2 \cdot \hat{G_1} \cdot \lambda_1$, it gives the USI-scheduler a large operating space to achieve the dynamic balance between the battery life and processing speed. The scheduler can set the b flow within a large operating interval. This interval defines the working operating zone for polymorphic computing. We note that if $P_1 \cdot \hat{G_2} \cdot \lambda_2 < P_2 \cdot \hat{G_1} \cdot \lambda_1$, then any polymorphic morphism decision will introduce both extra delay and extra energy. This leaves zero operating room for polymorphic computing. Under this condition, we should just use the fixed-morphism approach to harvest the potential of heterogeneous computing architecture.

2.1.4 The Queueing Model for Simulation

Section 2.1.3 demonstrates the theoretical operating zone of the USI-based scheduler and polymorphic computing. But this model has two flaws in practice: **(1)** a key model parameter, the unbalance coefficients λ are difficult to estimate and utilize for schedulers, **(2)** it does not model the waiting ready queue which is common in most OSs. The scheduler's decision should be based on a perceivable variable estimate of the work loads. There are no techniques for scheduler to detect the sudden changes in the threads' generation rates within short time windows (of the order of the OS scheduling period) in a real OS. So we extend the previous model by adding the "water tank" to model waiting queue, and using the height of water to infer the underlying parameters: λ. The expected waiting time of each thread is linear in its height (position) in the tank. The unbalances of the workloads can be observed by detecting the sudden changes in the height of the tank (waiting queue's length). Figure 2.2 (b) presents the relationship between the basic elements in the practical "tank" model. The scheduler can predict the expected delay of each thread based on its position in the queue, and then decide

the optimal size of valve (the dispatch rate of threads to achieve the maximum USI for the two queues). Instead of watching λ_1, λ_2, a real scheduler should watch the tank heights h_1, h_2, and change the scheduling decision and policy when sudden changes to h occur.

Using the tank model, we can obtain the general turnaround time change over all threads in the waiting queues in response to morphism change for one thread. Figure 2.3 demonstrates the effect of polymorphic computing on the queues. A large number (h_2) of threads wait at Queue-2 while a small number (h_1) of threads are waiting at Queue-1. We assume that all threads within a queue are already sorted according to their priority. A thread with low priority has no chance to execute at the corresponding computing core before all threads with higher priority in the same queue do.

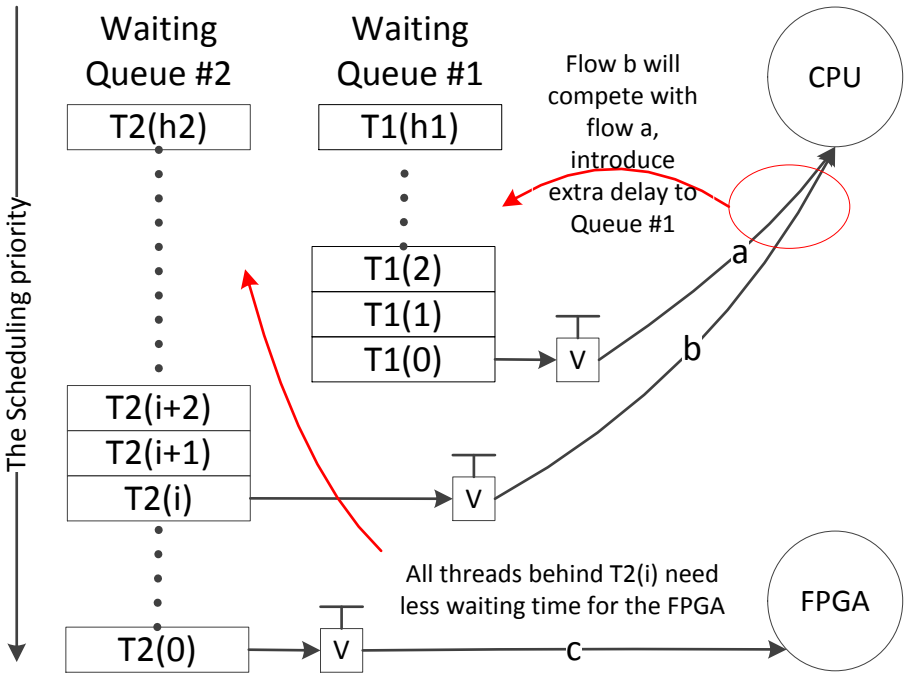

Figure 2.3 The Tank Model

Under the scenario that Type-2 threads' generation rate is higher than Type-1's rate, the CPU cores will have more vacancy compared with the FPGA tiles. Since there are many threads waiting in Queue-2, we can reduce the load on Queue-2 by utilizing the free CPU

cores. If we choose the thread T_i in Queue-2 and issue it to CPU cores immediately, then all the threads behind it in the Queue-2 $(T_{i+1}, T_{i+2}, ..., T_n)$ might be dispatched a little earlier. We should notice that the reduced time benefits are not guaranteed for every thread behind T_i because the threads' waiting time depends on many factors like inter-thread dependency and the valve's size change (scheduling decision). If the scheduler adopts a dynamic priority policy instead of static priority policy, then the order in the queue will change cycle by cycle. In such a case, the expected waiting time prediction is inaccurate. The speed benefits we discuss here are merely a heuristic guess of benefits. We should also note that threads in Queue-1 will suffer extra delay if we map T_i to CPU cores. Here we assume that there are n_1 same CPU cores and n_2 same FPGA cores in the system. We will estimate such decision's effect in the following equation:

For the thread T_i, if it continues to wait in Queue-2, its total life time is:

$$T_{execute} + T_{wait} = \frac{CPL_2}{P_2} + \frac{i \cdot C\hat{P}L_2}{n_2 \cdot P_2} \tag{2.3}$$

Here CPL (Computation Processing Length) captures the average computation needs of a thread in the units P's processing rate is specified. We assume that all the threads have the same computation needs CPL for a simplified model. CPL captures instantaneous value and $C\hat{P}L$ captures the mean of CPL based on some random distribution.

If we now issue thread T_i to CPU, its total life time is:

$$T_{execute} + T_{wait} = \frac{CPL_2}{P_1} + 0 \tag{2.4}$$

Each thread behind thread T_i will benefit by not having to wait on T_i, so the net benefit to delay in Queue 2 is:

$$(h_2 - i) \cdot \frac{CPL_2}{n_2 \cdot P_2} \tag{2.5}$$

However, all the threads in Queue 1 will suffer additional waiting time:

$$h_1 \cdot \frac{CPL_2}{n_1 \cdot P_1} \tag{2.6}$$

If we assume that the waiting queues 1, 2 are stable in their thread membership, then we can claim that the overall delay of all threads will be reduced if:

$$\frac{CPL_2}{P_2} + \frac{i \cdot C\hat{P}L_2}{n_2 \cdot P_2} > \frac{CPL_2}{P_1} - (h_2 - i) \cdot \frac{CPL_2}{n_2 \cdot P_2} + h_1 \cdot \frac{CPL_2}{n_1 \cdot P_1} \tag{2.7}$$

Solving Equation 2.7, we get:

$$n_1 P_1 h_2 C\hat{P}L_2 + n_2 n_1 P_1 CPL_2 > n_2 n_1 P_2 CPL_2 + n_2 P_2 h_1 CPL_2 \tag{2.8}$$

If we assume that $C\hat{P}L_2 = CPL_2$, then we get a simplified version:

$$n_1 P_1 h_2 + n_2 n_1 P_1 > n_2 n_1 P_2 + n_2 P_2 h_1 \tag{2.9}$$

We should note that Equation 2.9 indicates that no specific choice for T_i maximizes the delay improvement. This is because we have assumed $C\hat{P}L_2 = CPL_2$. The polymorphic computing-thread movement across types - brings speed benefits under certain steady-state system conditions specified in Equation 2.9. This is contradictory to the traditional expectation that the maximum throughput will bring overall speed improvement for the system [25, 21]. If we apply normal scheduler whose goal is to maximize utilization of computing resources by minimizing the cores' idle time, we will introduce extra delay over all threads in a polymorphic computing system under certain conditions.

2.2 USI-based Scheduling

2.2.1 The Proper USI-function

In our previous work [14], we use the sigmoid function to model user satisfaction as a function of processing throughput of the system. In this section, we will discuss the essential features and characteristics of USI functions. We derive one simple USI function for our simulations. Due to lack of psychological experimental data, we do not have any empirical proof to support our arguments about user satisfaction dependence on system throughput and battery level. We assume that a normal user would favor speed when battery level is high. Users favor energy over speed when battery level is low. We use a simple linear function to model this relationship:

$$\Delta USI/\Delta T \sim Battery\ Level/Maximum\ Battery\ Level \tag{2.10}$$

In order to simplify our simulations, we do not adopt the sigmoid function from [22]. Instead, we assume that the user satisfaction is linear in time:

$$\Delta USI/\Delta T \sim Turnaround\ Time/Reference\ Time \tag{2.11}$$

With this USI model, the application which has longer turnaround time will suffer larger USI loss than the application with shorter turnaround time even for the same marginal time penalty (or saving). $[(T_1 + \Delta T)/T_{ref}] > [(T_2 + \Delta T)/T_{ref}]$ for $T_1 > T_2$. This property does not hold for other USI functions, like a sigmoid function, where ΔUSI does not increase linearly with the turnaround time. Due to saturation of USI function with high T, some threads that have already waited for long time will see their marginal USI degradation close to 0. Hence they may further suffer by waiting even longer (possibly indefinitely). This trend will make the system unstable and generate some unreasonable results. Some threads will experience extremely long waiting time. Many functions possess the property/characteristics specified in Equations 2.10 and 2.11. We will use a very simple one in the rest of this paper:

$$USI_{apps} = f(T_{apps}, B_c) = 1 - \left(\frac{T_{apps}}{T_{ref}}\right)^2 \cdot \left(\frac{B_c}{B_{ref}}\right) \tag{2.12}$$

The USI function in Equation 2.12 calculates each application's USI under a specific battery level. T_{apps} is the turnaround time of the application; T_{ref} is the reference time for normalization. B_c is the current battery level; and B_i is the full/maximum (reference) battery level. This function is the simplest quadratic form that matches the properties in Equations 2.10 and 2.11.

2.2.2 The Influence of Non-linear Function

As the function in Equation 2.9 indicates, the choice of thread T_i to issue to Type-1 CPU morphism from the overloaded queue will not affect the overall improvement of delay. This claim is based on the assumption that generally shorter overall turnaround time of threads will lead to shorter overall turnaround time for the application. But this rule will not hold if we change the evaluation metric from delay to the overall USI of applications. We should note that if we change the evaluation metric from a function linear in time to a quadratic function

of time, the same ΔT improvement results in different USI improvement based on the current USI. The slope of a quadratic function contains a T term. Hence the polymorphic scheduling decisions are different. Figure 2.4 demonstrates the influence of a quadratic USI function.

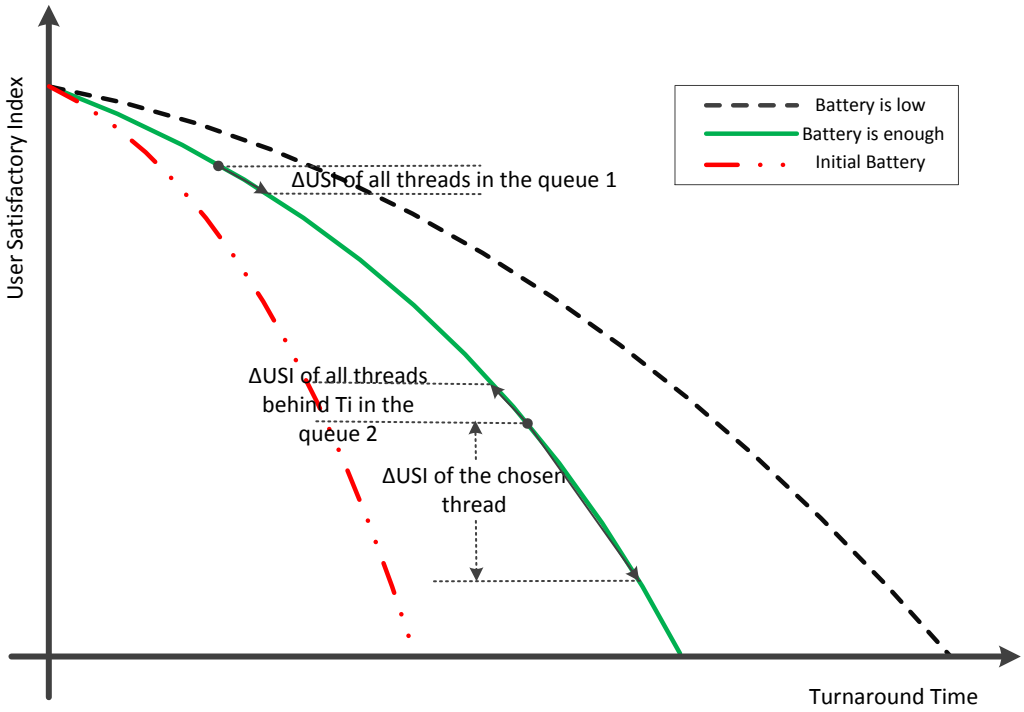

Figure 2.4 The Properties of the USI functions

Figure 2.4 presents several curves for USI under different battery level conditions. It demonstrates that a ΔT improvement for a thread that is already fast results in a smaller marginal USI gain compared to the same ΔT gain for a thread that is slow. We assume the basic scheduling policy to be FCFS (first come first serve) for simple mathematical analysis. Under the FCFS policy, a thread at low position means that it has already waited for a long time. So it does not need too much more time to be issued at the matched (better Energy-Delay profile) computing core. But if we issue it immediately to an un-matched core (worse energy-time profile), the compulsory longer execution delay may not be compensated for by the short skipped waiting delay. The quadratic USI function determines that this decision will introduce loss of USI. Since all the other threads get only a small benefit from this polymorphic switch

action, the accumulated small amounts of ΔUSI will not compensate the single ΔUSI loss of the chosen thread.

If we consider the problem at the application level, instead of at the thread level, then this problem will be more complex. We assume that an application is composed of several threads, and the dependences between threads form a directed acyclic graph (a thread control flow graph). Then some threads are in the critical path of the thread CFG, and some are not. If we change the morphism of a thread, whether the critical path of the corresponding application will change or not is hard to predict. The real critical path of the application depends on the current queue condition and position of each thread in the queues. The real critical path of the application may change due to scheduling decisions made in the future. Since an accurate mathematical method to predict the critical threads in the waiting queue is not available, we can only rely on the numerical model to evaluate the influence of each polymorphic decision.

2.2.3 Procedure for Calculating Polymorphic Decision's Influence

As the previous section states, if we adopt non-linear functions to model user satisfaction, then choice of thread to switch morphism does affect the QoS (overall USI) of the system. A procedure for calculating and predicting the effect of thread choice on the ΔUSI of the current applications in the waiting queue is presented here. In the Thread CFG, each thread node can be labeled with tag indicating the worst case path length to a leaf node (end of application computation). In order to perform application level prediction, the scheduler can utilize these tags to predict the length of the left-over computation. We should note that these tags only give a best-case schedule - assuming the best Energy-Delay profile morphism selection for all the threads down-stream, and with zero waiting time in the queues. This is fairly inaccurate because the final schedule of threads depends on scheduling decision made in the future. This kind of tags-based approach can only partly reflect threads' importance in the critical path of corresponding application's dependence graph. Another prediction could be based on history. At the first step, we use the uni-morphism policy like FCFS to form the current queue, and then we can predict the application's critical path based on the current conditions. For each thread (i_{th} position in the active waiting Queue-2), we can calculate an expected turnaround

path of this thread's corresponding application:

$$T_i = T_{wait} + T_{execute} + T_{rest} = \frac{i \cdot \hat{CPL}_2}{n_2 \cdot P_2} + \frac{CPL_2}{P_2} + Tag_i \tag{2.13}$$

In Equation 2.13, the T_{rest} is the predicted remaining path length (to a leaf) in the dependence graph of this thread. If we enumerate expected turnaround/termination paths for all current active threads of an application, we can find out the longest and the second longest termination paths. We will record these paths and mark corresponding threads' information. In the second step, we calculate out each thread's ΔT and corresponding ΔUSI of the application when we change this thread's morphism:

$$\Delta USI = f_{usi}(T_{new}) - f_{usi}(T_{critical}) \tag{2.14}$$

We can use the information obtained at Step 1 to roughly estimate whether the application's turnaround time will increase, decrease, or remain unchanged. If this thread is a critical thread of the application, then the application's T_{new} is:

$$\begin{cases} T_{in} = \frac{CPL_i}{P_1} + Tag_i, & \text{when } T_{in} > T_{second\_longest} \\ T_{second\_longest}, & \text{when } T_{in} < T_{second\_longest} \end{cases} \tag{2.15}$$

The T_{in} is the corresponding application's new expected turnaround/termination time. We obtain it using a method similar to the one in Equation 2.4. If this thread is not a critical thread, then T_{new}'s expression is:

$$\begin{cases} T_{in} = \frac{CPL_i}{P_1} + Tag_i, & \text{when } T_{in} > T_{critical} \\ T_{critical}, & \text{when } T_{in} < T_{critical} \end{cases} \tag{2.16}$$

The $T_{critical}$ is the recorded, longest turnaround time for the application. Equation 2.16 suggests if the thread selected for morphism switch is not a critical thread, then we will only account for the possible extra USI loss of the decision when the expected critical turnaround time of the application is prolonged. In Step 3, we will calculate the influence of current decision on other applications. We search all the threads behind the chosen one and check the tag space to see if that thread is critical for its application. If so, then we will add the USI improvements or losses to the total ΔUSI of current choice. We will also check and calculate possible ΔUSI of

threads in Queue-1 using a method similar to Equation 2.6. The only difference is that here we will account for the ΔUSI of only really critical threads of some applications. The final step is searching the ΔUSI table to get the highest ΔUSI improvement value. The corresponding thread is the optimal choice for morphism switch. This procedure needs $O(n^2)$ time to calculate overall ΔUSI from thread switch from Queue-2 to Queue 1. Although $O(n^2)$ is high compared to a general scheduling algorithm, we should notice that the scheduling algorithm's speed is not our primary concern in this paper.

2.2.4 The Greedy Scheduling Algorithm

Once we determine the optimal thread choice for morphism switch for polymorphic computing, we should make a decision on whether we really issue the chosen thread to the less suitable core or not. A simple heuristic could issue these threads to other core types as long as there exist threads that could bring overall USI benefits. But this method does not consider the battery term in the USI function. The ΔUSI of a decision for morphism switch of $thread_i$ can be expressed as:

$$\Delta USI_i = \Delta USIi_1 + \Delta USIi_2 + ... + \Delta USIi_n \tag{2.17}$$

$$= \frac{B_c}{B_{ref}} \cdot (\frac{T_{1o}^2 - T_{1n}^2}{T_{ref}^2} + \frac{T_{2o}^2 - T_{2n}^2}{T_{ref}^2} + ... + \frac{T_{no}^2 - T_{nn}^2}{T_{ref}^2}) \tag{2.18}$$

In Equation 2.17, $\Delta USIi_n$ is the n_{th} application's marginal USI gain through morphism switch for $thread_i$; T_{io} is $application_i$'s old turnaround time without the polymorphic computing, and T_{in} is $application_i$'s new turnaround time if we change $Thread_i$'s morphism. If we simply set a threshold that $\Delta USI_{max} > 0$, then the term B_c/B_{ref} will not have any effect at all. So here we set a threshold:

$$\Delta USI_{max} > \lambda \tag{2.19}$$

The λ is the constant threshold for ΔUSI gain in changing threads' morphism. When the current battery level is high enough, we may find many threads would be qualified candidates for morphism change. However when the current battery level is low, then the battery term

will decrease the possibility that even the threads bearing maximum ΔUSI will exceed the threshold. This greedy algorithm targeting maximum overall USI for all applications is simple enough for implementation.

CHAPTER 3. EXPERIMENT

3.1 Simulation Methodology

3.1.1 The Simulation Platform based on SystemC

We build a simulator using SystemC [10] and Matlab to evaluate our model and theory of polymorphic computing. SystemC provides complete solution for simulating concurrent events along with all the C's features for simulating the software's serial behavior. Thus it is an ideal tool for the evaluation of a polymorphic computing system. Figure 3.1 presents our model's structure. We use Matlab to generate the workloads that bear tags of ID, application ID,

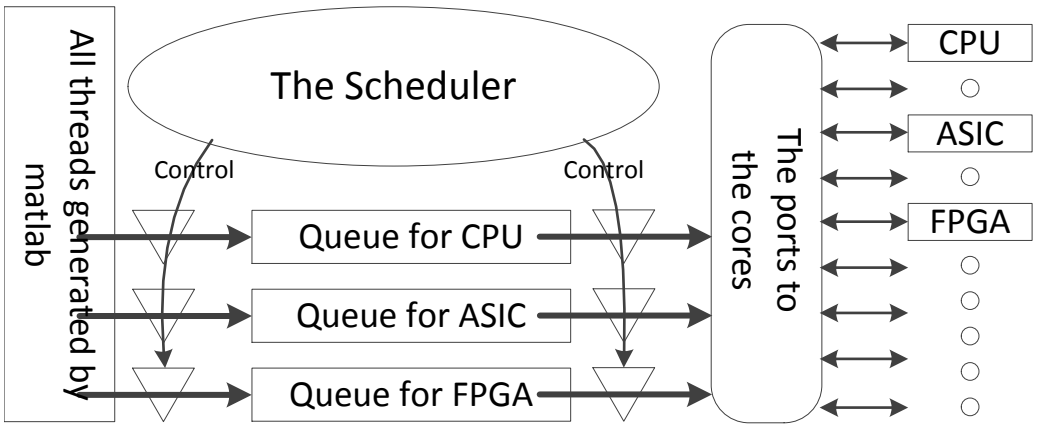

Figure 3.1 The Simulation Platform

dependence information, computing loads, other tags, and the most important, arrival time stamp. There is a long queue for simulating the hard disk of the polymorphic computing system. This queue just records the application generation in order and type. This allows for

various scheduling and polymorphic systems to be simulated with identical workloads. There are several fixed-size short queues for simulating the real scheduling windows. Each queue holds a specific kind of threads that are matched to the corresponding kind of cores. When a thread is generated by Matlab, all information will be stored in the first long queue (the hard disk model). The scheduling windows will only read in the thread with recorded arrival time stamp earlier than the current time stamp. This process emulates the user driven and external asynchronous event driven thread generation and their handling by the scheduler. When the thread generation rate is lower than the processing rate, the long queue will not affect the short queues at all. However, when the thread transient generation rate is higher than the processing rate of cores/queues, the scheduling windows are already overflowed, the scheduling window will read in a new thread only after the computing system finishes processing an old thread. The arrival time stamp of that new thread will be refreshed to current time stamp, which simulates the user driven throttling - negative feedback loop. Different types of computing cores are modeled as different count-down counters in this simulator. Each counter's rate which determines the computing speed and its corresponding power consumption are set according to Chung's paper [6]. The model of computing cores can be further expanded with complex behavior like interrupt, morphism changes, communicating with other cores, and other functions. The cores are connected with different scheduler with unified ports. The scheduler collects the status of the system through these ports and makes proper decisions based on the current conditions and the scheduling algorithm. The raw simulation results are also recorded by the scheduler, and are analyzed by Matlab.

3.1.2 The Model of Applications and Threads

In this research, computing is abstracted as a collection of applications. Each application in turn contains several threads. A thread is the minimum computing granularity mapped to a core. We use threads Dependence Flow Graph (DFG) to model the relationship between threads within one application. We assume that there is no dependency between applications. One thread may depend on another thread due to many reasons like necessary communication, data dependence, control dependence, etc. In the DFG all these relationships are abstracted

as a unidirectional edge between two nodes. The direction of the edge reflects the compulsory sequential order between two threads. Each node in the graph represents a thread. The topology of the edges and nodes is determined by the software's intrinsic structure. In the simulation we use two kinds of topologies to model the real-life software: the Embarrassingly Parallel Random (EPR) and the Random Tree(RT) [12]. Structure of these two topologies is shown in Figure 3.2. The applications with embarrassingly parallel topology contain several

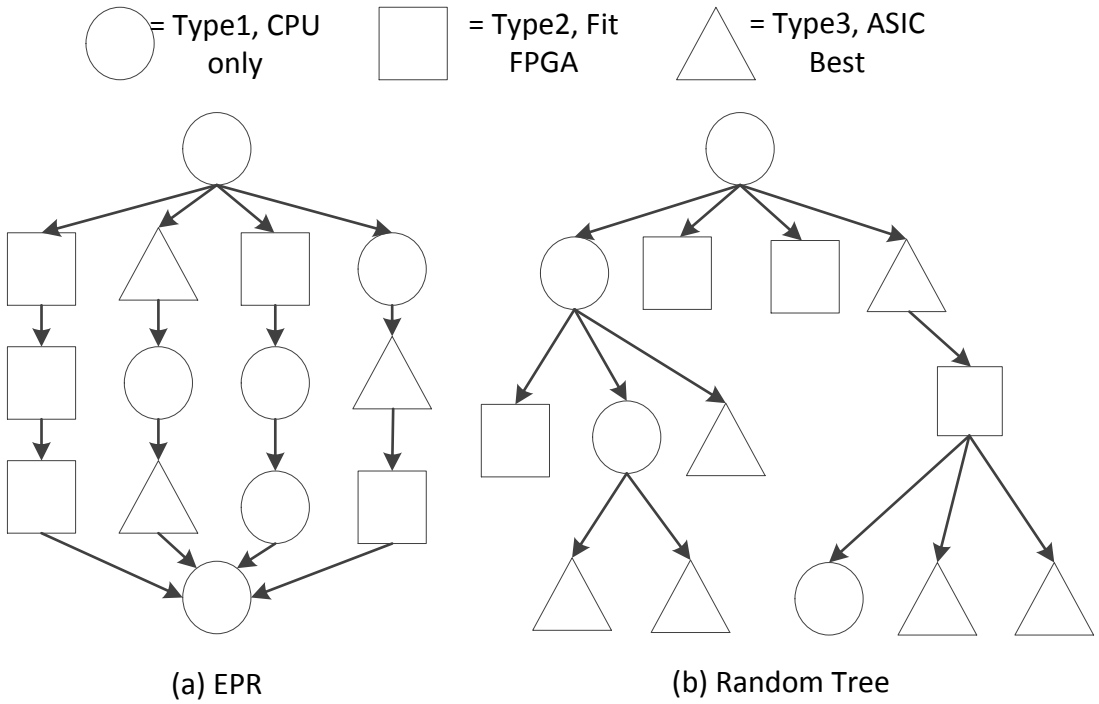

Figure 3.2 The structure of EPR and RT

parallel branches. These can be executed in a multi-processors system efficiently. The image processing program falls into this category. The tree structure is one of the classical topologies. Many algorithms and applications, like search, graph traversal fall into this category [12]. One key difference between these two topologies is that the EPR has an ending node while the RT does not. We use a random model during thread generation. The corresponding tags are also randomly generated. We generate different types of threads randomly with a pre-designated ratio. One important task in this work is to veryfy Equations 2.2 and 2.9. These

claim that a polymorphic computing platform's working operating zone strongly depends on the unbalance coefficient of different thread types. We adjust the random parameters in the generation process to test that Equation 2.2 applies over all kinds of topologies. We also evaluate our greedy polymorphic scheduling algorithm under various random parameters.

3.1.3 The Design of the Experiments

We have conducted two experiments to seek phenomenon that supports our theory and evaluate our proposed greedy algorithm presented in the previous sections. In the first experiment, the major goal is to test our working operating zone theory and the constraints in Equation 2.9. Three schedulers are evaluated in Experiment 1. All these three schedulers adopt the BFS as the basic scheduling policy for uni-morphism queue. The Linux BFS scheduler is based on the "virtual deadline first" policy. The "virtual deadline" policy is in essence a hybrid of FCFS and priority-based scheduling policy. In our simulation, we assign the threads a priority index to help the scheduler calculate the virtual deadline of the thread. A thread which is on the critical path of the application's DFG will receive higher priority (and have an earlier virtual deadline). The key difference between these three schedulers is their policy on polymorphic computing. One scheduler is designed to achieve maximum throughput. This scheduler will change the morphism of the thread at the FPGA queue head as long as its virtual deadline is later than the thread at the CPU queue head. Another scheduler is designed to achieve the maximum battery life. This scheduler will never change the morphism of threads, and achieve the maximum energy efficiency. The third scheduler will apply the constraint of Equation 2.9 to decide on polymorphic morphism switch. When different queues' length satisfy Equation 2.9, the third scheduler would change the FPGA queue head thread's morphism.

Experiment 2 is designed to evaluate a greedy algorithm to achieve maximum overall USI for all applications proposed in Section 2.2. Three other schedulers for polymorphic computing are evaluated in this experiment. The uni-morphism scheduling policies of these three schedulers are slightly different. Equation 2.12 indicates that a thread with longer turnaround time (high T) would suffer higher overall ΔUSI punishment compared to a thread with shorter time with

the same time penalty ΔT. Here we give the threads with higher predicted turnaround time higher priority in the basic uni-morphism scheduling policy. The first scheduler in Experiment 2 is designed to achieve the minimum overall turnaround time. This scheduler will change the morphism of threads in Queue-2 as long as it can foresee overall turnaround time improvements. The second scheduler in Experiment 2 is similar to the scheduler in Experiment 1 designed to achieve maximum energy efficiency. The third scheduler adopts the proposed greedy algorithm to achieve overall USI improvements.

3.2 The Simulation Results

3.2.1 Simulating Results for Verifying the Model

In Experiment 1, a simple multi-core model was adopted: there is only 1 CPU core and 1 FPGA core. Under this model, the degree of parallelism of the test-bench won't affect the simulation results much, since there is only one core for execution of each type of threads. We will adopt the same model in Experiment 2. We will extend our simulator with complex cores organization in our future work. We change the unbalance coefficients of CPU threads and FPGA threads to simulate the fluctuations in incoming workloads. Seven scenarios of combination of unbalance coefficients were tested: 0.2(CPU):1.8(FPGA); 0.4 : 1.6; 0.6 : 1.4; 0.8 : 1.2; 1.0 : 1.0, 0.75 : 0.75; 0.5 : 0.5. Under the ratio of 1.0 : 1.0, the incoming workloads are balanced with the corresponding core's processing ability . We test both kinds of application topologies mentioned in section 3.1.2. In the rest of this paper, we will use the abbreviation ENG to refer to the scheduler designed to achieve maximum energy efficiency; the abbreviation THR to refer to the scheduler designed to achieve the maximum throughput; and the abbreviation DEL to refer to the scheduler designed to achieve the minimum overall turnaround time. The original data of Experiment-1 are given at Table 3.1,3.2,3.3, 3.4. In these tables, all data are counted in unit of time of the simulator. The Table 3.1, 3.2 present the battery life of the systems using three different schedulers. The workloads is modeled in EPR and RT topologies respectively. And the Table 3.3, 3.4 present the average turnaround time of all given applications (in EPR and RT topologies) under three different schedulers.

Table 3.1 The Battery Life in Experiment 1(EPR Topology)

Ratios	THR	DEL	ENG
0.2:1.8	76150	126700	175550
0.4:1.6	76270	110100	142220
0.6:1.4	76770	96110	115300
0.8:1.2	78950	84270	91800
1.0:1.0	81740	78770	79840
0.75:0.75	81520	79000	80130
0.5:0.5	81590	78620	79280

Table 3.2 The Battery Life in Experiment 1(RT Topology)

Ratios	THR	DEL	ENG
0.2:1.8	76330	99500	168360
0.4:1.6	76330	93280	139670
0.6:1.4	76850	86970	114600
0.8:1.2	78860	82090	94770
1.0:1.0	82120	80030	84570
0.75:0.75	82100	80620	85110
0.5:0.5	82320	80440	84780

In the rest of this paper, Figures 3.3, 3.4, 3.5, 3.6, 3.7 - (a) presents the results under the test bench in which the applications' DFG is the EPR topology and igures 3.3, 3.4, 3.5, 3.6, 3.7 - (b) presents the results under the test bench in which the applications' DFG is the RT topology. Since the unit of time in simulator does not contain any physical meaning, we normalize the results to make the comparison more clear. In the figure, all the results of the battery life and turnaround time of applications of THR and DEL schedulers are normalized to the simulation results of the ENG scheduler under the same scenario of parameters setting.

Table 3.3 The Average Turnaround Time in Experiment 1(EPR Topology)

Ratios	THR	DEL	ENG
0.2:1.8	180.6562	198.4825	185.8586
0.4:1.6	206.0369	204.4091	193.5645
0.6:1.4	262.9142	210.9041	204.6841
0.8:1.2	344.7477	225.6985	221.5169
1.0:1.0	404.4615	291.3241	291.0529
0.75:0.75	395.21	292.978	292.9172
0.5:0.5	404.9379	292.3849	292.6922

Table 3.4 The Average Turnaround Time in Experiment 1(RT Topology)

Ratios	THR	DEL	ENG
0.2:1.8	172.802	197.0531	178.4907
0.4:1.6	193.543	204.9054	188.5864
0.6:1.4	236.6603	220.7572	209.6983
0.8:1.2	306.2964	245.4203	237.4565
1.0:1.0	400.8124	305.5234	305.9304
0.75:0.75	391.3132	305.1419	306.2073
0.5:0.5	400.2689	307.4564	308.898

From Figure 3.3, we note that the battery life difference between the THR/ENG and DEL/ENG are linear in the unbalance coefficients' difference. When the incoming threads flows are very uneven, scheduling results of DEL and ENG have great differences. These two schedulers define the boundary of the effective operating zone of the scheduler on polymorphic computing. When the incoming threads flows are well balanced, the polymorphic scheduler cannot generate result any different than the traditional scheduler. This phenomenon supports the theory that the polymorphic system's efficient operating zone is linear in the difference of incoming rates between different types of threads. The simulation results of the last three scenarios indicate that when the incoming threads are well balanced, the polymorphic computing does not contribute at all.

Figure 3.4 presents the overall turnaround time of different schedulers under various ratios of incoming thread types. We should note that the THR scheduler suffers great overall delay slowdown when the incoming rate are balanced. The average turnaround time of the applications in DEL scheduler is always shorter than the ENG scheduler. Since the scheduler DEL adopts the Equation 2.9 as the critical threshold for decision on morphism changing, the comparison supports the Equation 2.9. The comparison of the average turnaround time of THR and ENG scheduler is also contradicts the conventional opinion that maximum system throughput would bring minimum overall turnaround time of applications [25], [21]. Besides, the linear relationship between the overall turnaround time improvements of DEL scheduler compared to ENG scheduler and the imbalanced ratio also supports our theory about effective operating zone of polymorphic computing system.

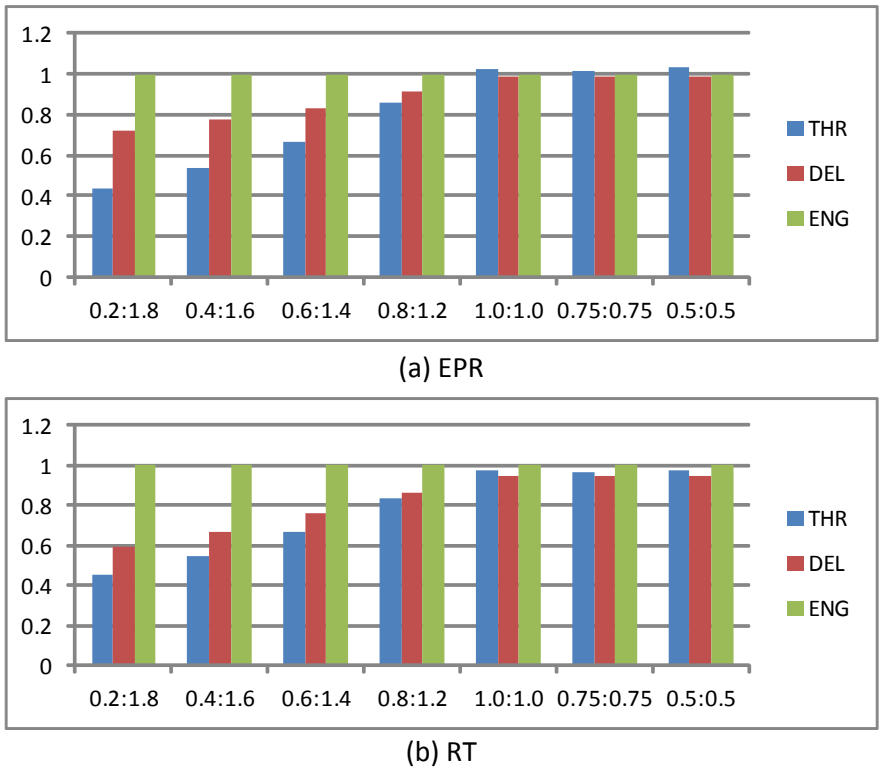

Figure 3.3 The Normalized Battery Life in Experiment 1

3.2.2 Evaluation of the Greedy Algorithm

In Experiment 2, we adopt test method similar to Experiment 1 to evaluate the greedy algorithm proposed in section 2.2. In the rest of this paper, we will use the abbreviation MEE to refer to the scheduler designed to achieve maximum energy efficiency; the abbreviation MT to refer to the scheduler designed to achieve the minimum overall turnaround time of applications; and the abbreviation GRE to refer to the greedy scheduler designed to achieve the maximum USI. We will first present the performance of these three schedulers using the traditional metrics like the average turnaround time of applications. We will then measure the performance of the schedulers using our proposed metric, the average USI of the applications. In order to calculate the average USI in a fair way, we assume that all schedulers are given the identical set of applications, and the scheduler that fail to finish any applications due to insufficient battery life will receive an individual 0 score of USI for those applications. We

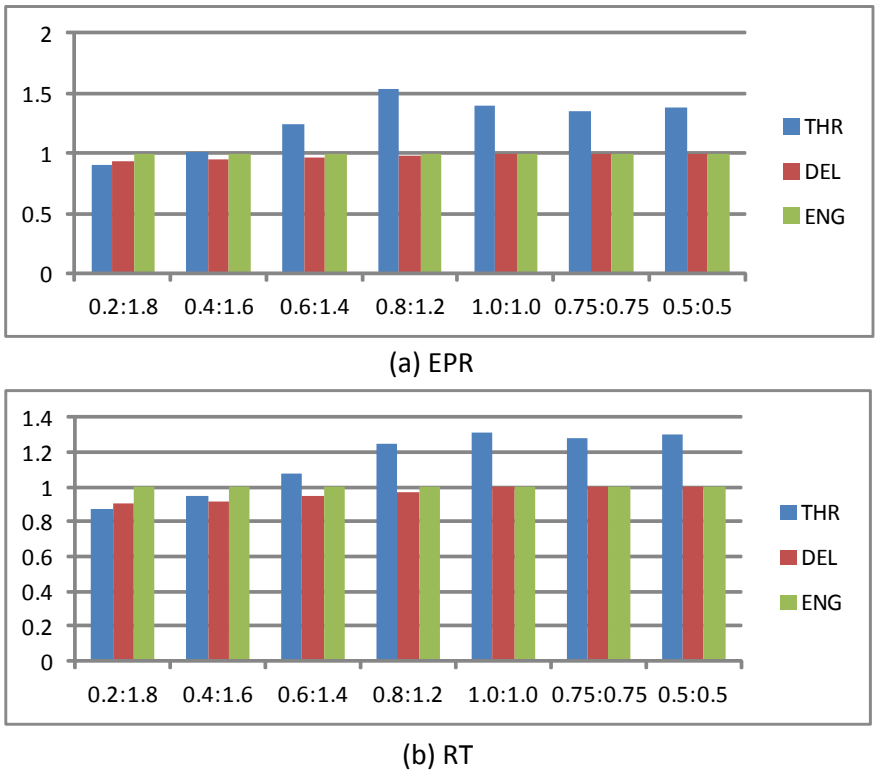

Figure 3.4 The Normalized Turnaround Time in Experiment 1

also set a threshold for the turnaround time of the applications similar to what iOS does [1]. If an application fails to respond within 5ms in iOS, it is terminated. We count all the applications that fail to meet the deadline. We will use these counts as another measurement of the scheduler's performance. A user should not be made unhappy/dis-satisfied too frequently. The raw data is given at the Table 3.5, 3.6, 3.7, 3.8, 3.9, 3.10. The Table 3.5, 3.6 present the average turnaround time of given applications under different scheduler. The Table 3.7, 3.8 present the average USI of given applications, and the Table 3.9, 3.10 present the counts of applications that mismatch the settled deadline.

We adopt the approach that similar to Experiment 1 to deal the raw data, normalization, to make the trend more clear. All the results of the average turnaround time of applications of MT and GRE schedulers are normalized to the simulation results of the MEE scheduler under the same scenario of parameters setting. Figure 3.5 presents the normalized average turnaround

Table 3.5 The Average Turnaround Time of Applications in Experiment 2(EPR Topology)

Ratios	MT	GRE	MEE
0.2:1.8	549.5789	658.0209	694.3015
0.4:1.6	582.5089	632.8593	676.2429
0.6:1.4	532.2711	573.8028	584.8117
0.8:1.2	522.827	528.2496	535.6558
1.0:1.0	554.1447	535.3327	534.489
0.5:0.5	426.03	414.0762	413.3046

Table 3.6 The Average Turnaround Time of Applications in Experiment 2 (RT Topology)

Ratios	MT	GRE	MEE
0.2:1.8	443.5982	567.0232	595.8375
0.4:1.6	450.2102	526.7945	546.7762
0.6:1.4	405.7101	448.0000	470.2008
0.8:1.2	397.3284	409.9574	414.0548
1.0:1.0	428.475	412.5551	411.1509
0.5:0.5	356.8667	344.9772	344.5264

time of the finished applications. The MT scheduler achieves significant reduction in average turnaround time compared with the MEE and GRE schedulers when the incoming ratio is not balanced. It may prolong the application's turnaround time when the ratio is balanced. The average turnaround time of GRE scheduler is always shorter than that of MEE scheduler. The improvement of GRE scheduler is not as significant as the MT scheduler.

Figure 3.6 presents the average USI of three schedulers. The USI is calculated according to Equation 2.12 and averaged over all applications. The scheduler with higher average USI is better. We should note that under this metric, the MT scheduler is much worse than the other two. Main reason is that the MT scheduler sacrifices too much energy to achieve the

Table 3.7 The Average USI in Experiment 2(EPR Topology)

Ratios	MT	GRE	MEE
0.2:1.8	0.450885456	0.700596497	0.738562951
0.4:1.6	0.505019605	0.715592355	0.742620099
0.6:1.4	0.616740438	0.791685693	0.806407153
0.8:1.2	0.723480895	0.82474148	0.832233046
1.0:1.0	0.787936273	0.832280561	0.832455311
0.5:0.5	0.929951021	0.999621917	0.907435271

Table 3.8 The Average USI in Experiment 2(RT Topology)

Ratios	MT	GRE	MEE
0.2:1.8	0.486470268	0.779889891	0.825682061
0.4:1.6	0.554277278	0.797297809	0.843242445
0.6:1.4	0.679365574	0.856055055	0.885508743
0.8:1.2	0.79227343	0.901201932	0.909910467
1.0:1.0	0.857810943	0.907495472	0.915477358
0.5:0.5	0.877652075	0.935950566	0.943640755

Table 3.9 The Applications Counts that Overdue in Experiment 2(EPR Topology)

Ratios	MT	GRE	MEE
0.2:1.8	0	26	119
0.4:1.6	10	12	68
0.6:1.4	13	10	18
0.8:1.2	7	6	13
1.0:1.0	5	2	2
0.5:0.5	0	0	0

significant turnaround time improvement shown in Figure 3.5. Many applications cannot finish under the given battery volume budget if we using the MT scheduler. The GRE scheduler also sacrifices energy to lower the average turnaround time, but it utilizes the greedy algorithm to guarantee the QoS, so its average USI is very close to that of the MEE scheduler. We have to admit that current definition of USI does not favor the GRE scheduler too much. One possible reason is that current definition weight the finished application too heavy. Any sacrificing of energy to improve the overall turnaround time will lead to loss of application and great loss of USI. So the traditional fixed mapping scheduler(MEE scheduler) is over-rated under current USI definition. Modifying the definition of USI and solving this problem is one of our future

Table 3.10 The Applications Counts that Overdue in Experiment 2(RT Topology)

Ratios	MT	GRE	MEE
0.2:1.8	2	34	71
0.4:1.6	0	5	42
0.6:1.4	0	0	4
0.8:1.2	0	0	0
1.0:1.0	0	0	0
0.5:0.5	0	0	0

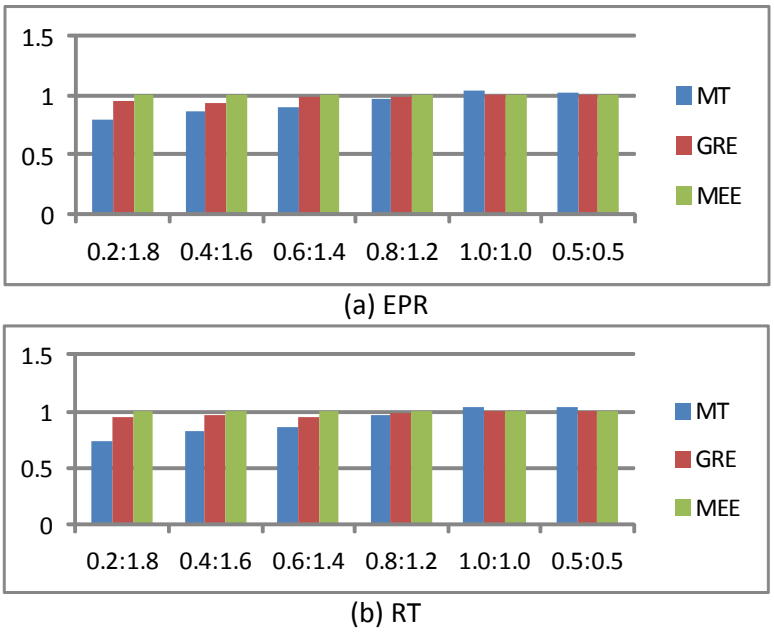

Figure 3.5 The Normalized Turnaround Time in Experiment 2

research direction. Figure 3.7 presents the applications counts whose turnaround time exceeds the threshold.

The count of MEE scheduler is significantly higher than the other two schedulers. This means its service is bad from the user satisfaction point of view. The counts of MT and GRE schedulers are close to each other. From the comprehensive point of view, the GRE scheduler achieves the compromised trade-off between the battery life and average turnaround time.

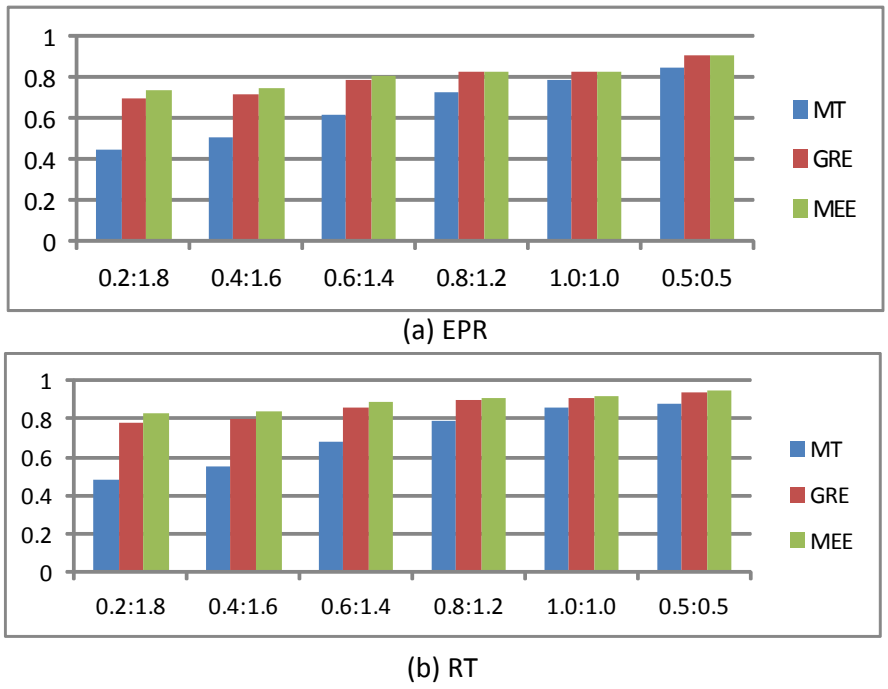

Figure 3.6 The Average USI for all given applications

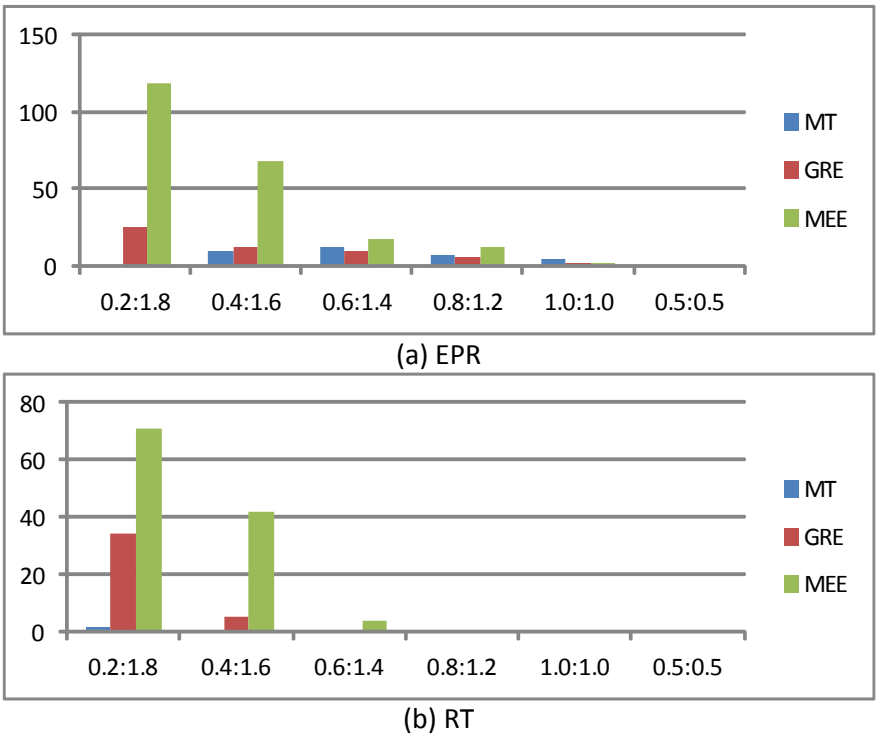

Figure 3.7 The Counts of Overdue Applications

CHAPTER 4. CONCLUSIONS AND FUTURE WORK

4.1 Conclusions

This paper proposes polymorphic computing system which consists of architecturally diverse cores - CPU, GPU, FPGA, and ASIC. Polymorphic software architecture in which threads are designed for multiple core architectures exploits the best features of a polymorphic computing system. This paper develops various theoretical models to understand when it might be beneficial to use polymorphic software on a polymorphic system. Many of these theoretical models are parameterized by empirical parameters. We built a simulation environment to emulate a polymorphic system along a broad choice of system parameters to validate our theoretical models.

We start with a simple queuing model to analyze the polymorphic computing system. We analyze the design space for a polymorphic system and present a theoretical analysis of the viable working operating zone of this system. The USI based scheduling is also covered in this analysis. A novel greedy scheduler designed to achieve maximum USI is also proposed. A SystemC based simulator for polymorphic computing is built. The simulation results validate our theory. A greedy scheduling algorithm is also evaluated in this simulator. The scheduler performs the trade-off between speed and energy. Our work indicates that the USI-based polymorphic scheduling on heterogeneous system is a good software solution to match future embedded system's requirements.

4.2 Future Work

Various interesting topics are also raised in this work, like, what will happen if we extend the model of heterogeneous cores with more types of cores like GPU, ASIC? Does the polymorphic

computing improve the system potential parallel processing ability under the massive multi-core system? Will the current theory and conclusions change, if we adopt the asymmetric thread-matching model? What new will occur if we consider the reconfigurable cores' reconfiguring time and energy cost? We will focus on discussing the last two questions in the rest of this section.

4.2.1 The asymmetric thread matching model

Among the basic assumptions of the model in this thesis, several are made for simplicity, and they are far away from the real environments. One is that the FPGA is better than CPU in both processing time and energy efficiency at a fixed rate for all the proper kinds of threads. In the Chung's paper [6], a general rate is given base on statistic average. We can see that the ratios of speed and energy efficiency between the CPU, FPGA, GPU and AISC are quite different for various kinds of testbench. If we introduce this feature to our model, then we would observe that CPU may gain the "comparative advantage" over the FPGA even it is inferior to FPGA for all kinds of threads. We borrow the term "comparative advantage" from the economic science where it is claimed that a developing country has the advantage of producing products in a higher cost than the developed country because the developed country are too busy to produce such products. Let's assume there are two subtypes of Type-2 threads, $subtype_A$ and $subtype_B$. The FPGA will gain 10 times of speed and energy efficiency than CPU for $subtype_A$ while gain only 2 times for $subtype_B$. Then there would be scenarios that FPGA cores are too busy for processing $subtype_A$ threads so the scheduler have to give some and only some $subtype_B$ threads to CPU because of the "comparative advantage" of CPU cores under such conditions. We can also borrow some well established theory from economic science to help analysing the new feature. But combining the "comparative advantage" to the USI functions is quite hard and we are still working on that. Numerical calculation in scheduler can not find this fundamental feature of threads. Another major obstacle is lacking statistic data to model such scenarios in simulator. Simulation with arbitrary setting is less persuasive. We should notice that the working zone of polymorphic computing will be enlarger if we consider the "comparative advantage" of the CPU.

After introducing the asymmetric mapping model, more complex model is possible. Previously, we assume that there are some threads that are unavailable for running on FPGA. This is a necessary simplified restriction under the symmetric thread-mapping model. If the FPGA can outperform the CPU for all the tasks at a fixed rate, then we won't put any CPU cores on the chip when designing the hardware of SOC chips. However, under the asymmetric mapping model, We could adopt the DeHon's statistic results [9] and enabling the type 1 threads to be mapped on FPGA. But here we will assert that FPGA will take more time and consume more energy for the type 1 threads. Then normally, we won't map type 1 threads to FPGA unless the CPU is too busy and FPGA cores are free. The Figure 4.1 (a) presents the asymmetric flow mapping model, and the Figure 4.1 (b) presents one numerical cost table as a case of this assumption. In the table the "T" means the time cost and "E" means the energy cost. We can extend this table with arbitrary number of processing cores and types of threads.

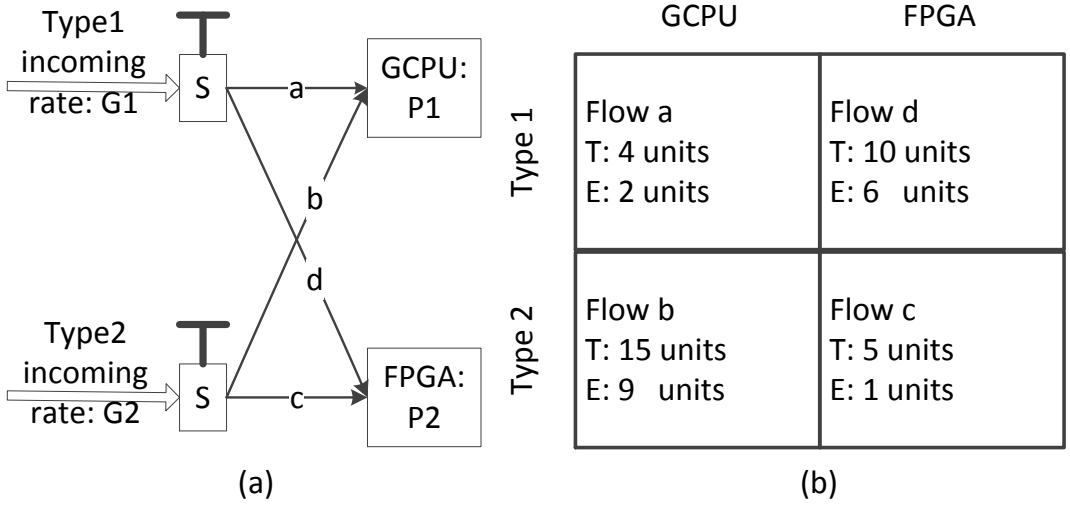

Figure 4.1 The Asymmetric Flow Mapping Model

Obviously, theoretical analysing or experimental results for such model are much harder to get compared with previous simplified models. Although modifying our current simulation model to adapt the asymmetric mapping assumption is pretty easy, we haven't conduct a experiment on such new model mainly due to lack of sound parameters setting. Intuitively,

adding this assumption will also enlarge the working zone of polymorphic computing, that extra energy and speed may be harvested by polymorphic computing even when CPU is too busy. Generally speaking, after considering the asymmetric mapping assumption which is more close to real environments, we can give the polymorphic computing system a further positive expectation due to the larger working zone .

4.2.2 Adding the cost model of reconfiguring

One interesting component in the polymorphic computing system is the FPGA, the dynamic reconfigurable hardware. In the polymorphic computing, we assume that the scheduler can change the configuring bits of FPGA tiles anytime. This function involves extra consumption of time and energy. In the final numerical results of chung's paper, the reconfiguring cost is counted in, but the details of reconfiguring frequency is not revealed in that paper. Obviously, the actual cost is linear to the real shift frequency decided by the scheduler. Then the scheduler's decision will change the time and energy consumption of the given workloads. We will illustrate the importance of the scheduler's decision on a extreme imaginary case stated below. We assume that there are two subtypes($subtype_A$ and $subtype_B$) of Type-2 threads in the original symmetric flow-mapping model, and we need to reconfigure the whole FPGA core every time that the configuring shifts from $subtype_A$ to $subtype_B$ or reversely. Here we assume that there is only one FPGA core in the system, and the incoming threads flow is just two subtypes of Type-2 threads alternatively. If we mapping all the incoming Type-2 threads to the only FPGA core, then we have to reconfigure the FPGA for every incoming thread. Let's assume the $subtype_B$ threads are not on the critical path of the applications' DFG; and they are pretty small, that mapping them to CPU core won't waste too much energy. If the scheduler map all the $subtype_B$ threads to CPU cores, then we can save all the reconfiguring time and energy cost. There may be chances that the saved reconfiguring energy and delay are lower than the extra energy and delay of $subtype_B$ threads on the CPU. After adding the reconfiguring cost, all the formulas for calculating ΔUSI in previous section will change. Definitely, the behavior of the scheduler will change due to this extra model. This problem will more complex if we add the multi-core model to the FPGA clusters. Previously, the scheduler can choose any vacant

FPGA to the next chosen Type-2 threads; now waiting longer for another configured tile may be a better decision for the next chosen threads if we consider the reconfiguring cost. Generally, we believe that adding the cost of reconfiguring will enlarge the working zone of polymorphic computing, and adding more interesting questions to the problem of scheduling on polymorphic computing system.

BIBLIOGRAPHY

[1] Apple, I. (June 2010). ios application programming guide.

[2] Borkar, S. (2007). Thousand core chips: a technology perspective. pages 746–749.

[3] Bower, F. A., Sorin, D. J., and Cox, L. P. (2008). The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28:17–25.

[4] Cardoso, J., Diniz, P., and Weinhardt, M. (2010). Compiling for reconfigurable computing: A survey. *ACM Computing Surveys (CSUR)*, 42(4):13.

[5] Cassidy, A. S. and Andreou, A. G. (2012). Beyond amdahl's law: An objective function that links multiprocessor performance gains to delay and energy. *Computers, IEEE Transactions on*, 61(8):1110 –1126.

[6] Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K. (2010). Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? pages 225–236.

[7] Curreri, J., Koehler, S., Holland, B., and George, A. (2008). Performance analysis with high-level languages for high-performance reconfigurable computing. pages 23 –30.

[8] Dally, W., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R., Parikh, V., Park, J., and Sheffield, D. (2008). Efficient embedded computing. *Computer*, 41(7):27 –32.

[9] DeHon, A. (2000). The density advantage of configurable computing. *Computer*, 33(4):41 –49.

[10] Ghosh, A., Tjiang, S., and Chandra, R. (2001). System modeling with systemc. pages 18 –20.

[11] Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B. C., Richardson, S., Kozyrakis, C., and Horowitz, M. (2010). Understanding sources of inefficiency in general-purpose chips. pages 37–47.

[12] He, Y., Liu, J., and Sun, H. (2011). Scheduling functionally heterogeneous systems with utilization balancing. pages 1187 –1198.

[13] Hill, M. and Marty, M. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38.

[14] Krishnamurthy, V., Ponpandi, S., and Tyagi, A. (2011). A novel thread scheduler design for polymorphic embedded systems. pages 75 –84.

[15] Ma, N., Lu, Z., Pang, Z., and Zheng, L. (2010). System-level exploration of mesh-based noc architectures for multimedia applications. pages 99 –104.

[16] Merritt, R. (2010). Smartphones can replace pcs. *EE Times*, 41.

[17] Moncrieff, D., Overill, R., and Wilson, S. (1996). Heterogeneous computing machines and amdahl's law. *Parallel Computing*, 22(3):407–413.

[18] Pal, S., Chatterjee, M., and Das, S. K. (2005). A two-level resource management scheme in wireless networks based on user-satisfaction. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):4–14.

[19] Pande, P., Grecu, C., Ivanov, A., Saleh, R., and De Micheli, G. (2005). Design, synthesis, and test of networks on chips. *Design Test of Computers, IEEE*, 22(5):404 – 413.

[20] Paul, J. and Meyer, B. (2007). Amdahls law revisited for single chip systems. *International Journal of Parallel Programming*, 35(2):101–123.

[21] Ramarao, P. and Tyagi, A. (2003). An adiabatic framework for a low energy mu;-architecture and compiler. pages 65 – 72.

[22] Shye, A., Ozisikyilmaz, B., Mallik, A., Memik, G., Dinda, P., Dick, R., and Choudhary, A. (2008). Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction. 36(3):427–438.

[23] Stamoulis, G., Kalopsikakis, D., and Kyrikoglou, A. (1999). Efficient agent-based negotiation for telecommunications services. 3:1989–1996.

[24] Taylor Groves, Jeff Knockel, E. S. (2009). Bfs vs. cfs - scheduler comparison.

[25] Tyagi, A. (1992). A principle of least computational action (preliminary version). pages 262 –266.

[26] Xiao, M., Shroff, N., and Chong, E. (2001). Utility-based power control in cellular wireless systems. 1:412–421.